Practical 1

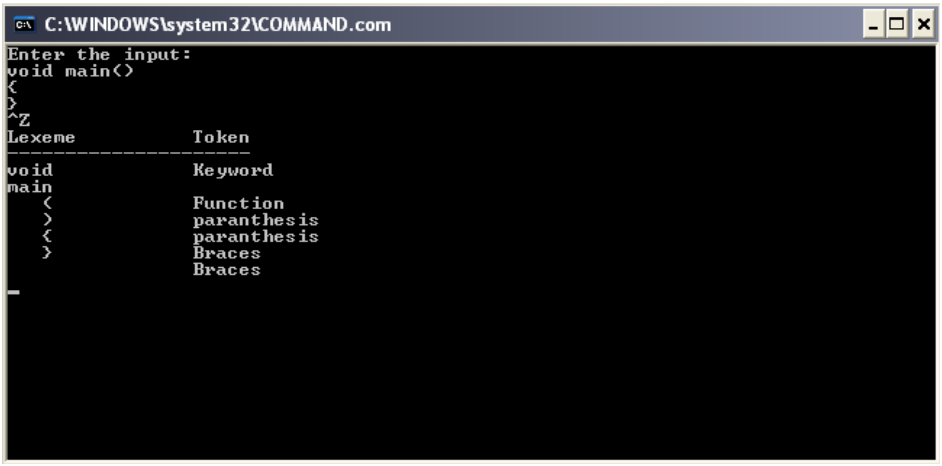
AIM: Write a Program for token specification and recognition. For example, Separation of keywords from identifiers, identify a set of operators etc. (Using File)

Source Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
 int i,j,k,p,c;
 char s[120],r[100];
 char par[6]={'(',')','{','}','[',']'};
 char sym[9]={'.',';',':',',','<','>','?','$','#'};
 char key[10]
[10]={"main","if","else","switch","void","do","while","for","return","printf"};
 char dat[4][10]={"int","float","char","double"};
 char opr[5]={'*','+','-','/','^'};
 FILE *fp;
 clrscr();
 printf("\n\n\t enter the file name");
 scanf("%s",s);
 fp=fopen(s,"r");
 c=0;
 do
 {  fscanf(fp,"%s",r);
  getch();
  for(i=0;i<6;i++)
   if(strchr(r,par[i])!=NULL)
    printf("\n paranthesis :%c",par[i]);
  for(i=0;i<9;i++)
   if(strchr(r,sym[i])!=NULL)
    printf("\n symbol :%c",sym[i]);
  for(i=0;i<10;i++)
   if(strstr(r,key[i])!=NULL)
    printf("\n keyword :%s",key[i]);
  for(i=0;i<4;i++)
   if((strstr(r,dat[i])&&(!strstr(r,"printf")))!=NULL)
    {
     printf("\n data type :%s",dat[i]);
     fscanf(fp,"%s",r);
     printf("\n identifiers :%s",r);
    }
  for(i=0;i<5;i++)
   if(strchr(r,opr[i])!=NULL)
    printf("\n operator :%c",opr[i]);
   p=c;
   c=ftell(fp);
 }
 while(p!=c);
 return 0;
```

}

OUTPUT:


C:\WINDOWS\system32\COMMAND.com
Enter the input:
void main()
{
}
^Z
Lexeme Token
void Keyword
main
(Function
) paranthesis
{ paranthesis
} Braces
Braces

Practical 2

AIM : Introduction to scanner generator tool: LEX.
LEX program to generate tokens for identifiers, numbers etc.

Ans :

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Source Code:

```
%{

        /* defination of menifest constants LL,LT,EQ,NE,GT,IF,THEN,ID,NUMBER,RELOOP*/
%}

/*   Regular Defination */

        delim  [\ + ln]
        use    { delim }*
        letter  [ A-Za-z ]
        digit   [ 0-9 ]
        id    { letter }( { letter } | { digit } )*
        number  { digit }*( \.{ digit }* )( E [ +|- ] { digit }* )
%%

        {ws}   { /* no action and no return */ }
        if          {printf("IF");}
```

```
then      {printf("THEN");}
else      {printf("ELSE");}
{id}      {printf("IDENTIFICATION");}
{number} {printf("NUMBER");}
"<"       {printf("LT");}
"<="       {printf("LE");}
"="       {printf("EQ");}
"<>"       {printf("NE");}
">"       {printf("GT");}
">="       {printf("GE");}

%%

main()
{
          yylex():
}
yywcpp()   { }
yyerror(char *s)
{
          char *S;
          printf("%s",s);
}
```

OUTPUT:

if (2<3)

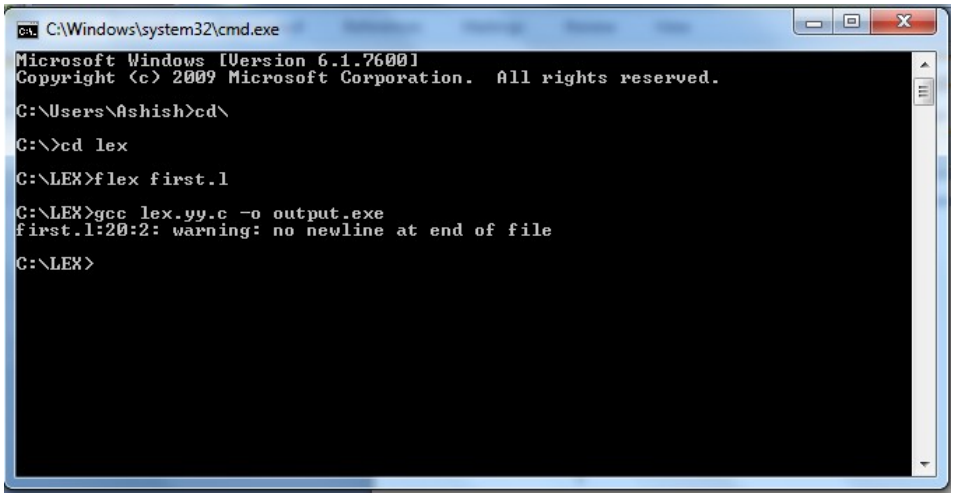
IF NUMBER LT NUMBER

Practical 3

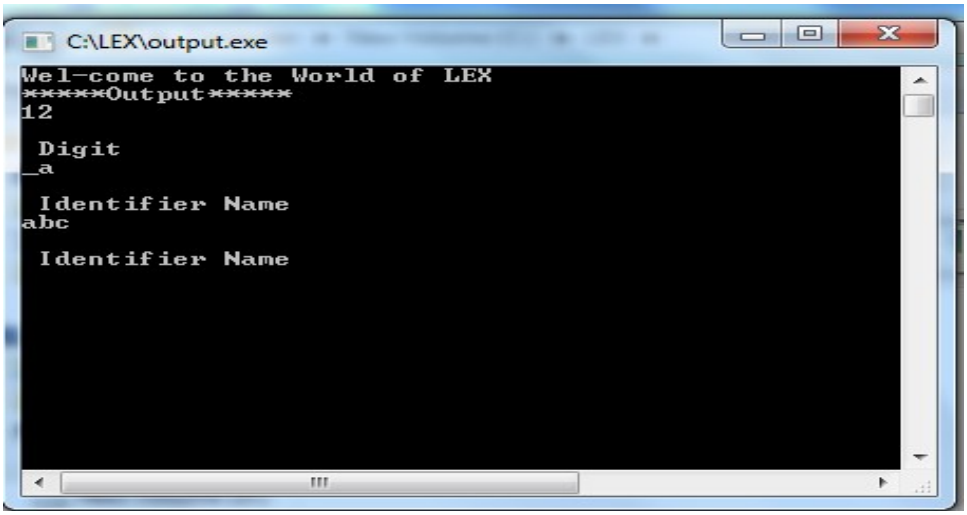
AIM: Write a LEX program to generate tokens for identifiers, numbers etc.

```
%{
#include<stdio.h>
%}
%%
[0-9]+ printf("\n Digit");
[a-zA-Z_][a-zA-Z_0-9]*     printf("\n Identifier Name");
%%
int yywrap()
{
return 1;
}
main()
{
   yylex();
}
```

OUTPUT:



➔ **Goto the folder and run the executable file output.exe.**

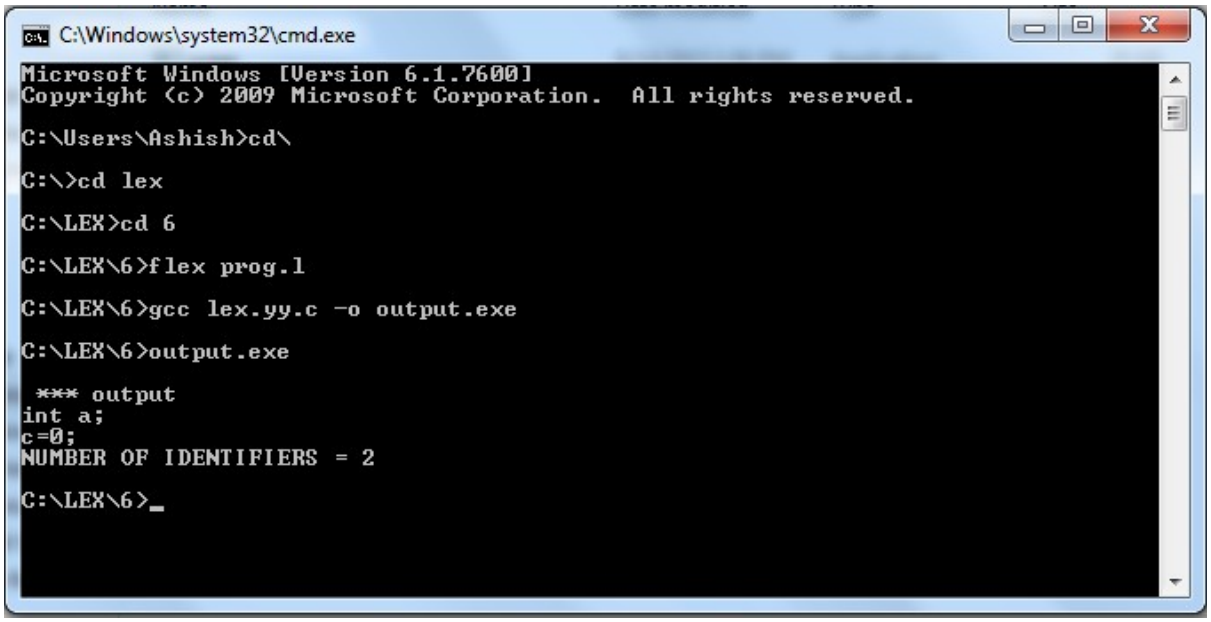

C:\LEX\output.exe
Wel-come to the World of LEX
*****Output*****
12
Digit
_a
Identifier Name
abc
Identifier Name

Practical-4

AIM: Write a LEX program to find the type of numbers.

```
%{
#include<stdio.h>
int id=0,flag=0;
%}
%%
"int"|"char"|"float"|"double" { flag=1; printf("%s",yytext); }
 ";" { flag=0;printf("%s",yytext); }
 [a-zA-Z][a-zA-z0-9]* { if(flag!=0) id++; printf("%s",yytext); }
  [a-zA-Z0-9]*"="[0-9]+ { id++; printf("%s",yytext); }
 [0] return(0);
 %%
 main()
{
  printf("\n *** output\n");
  yyin=fopen("f1.l","r");
   yylex();
  printf("\nNUMBER OF IDENTIFIERS = %d\n",id);
  fclose(yyin);
 }
   int yywrap()
 {
  return(1);
 }
```

OUTPUT :

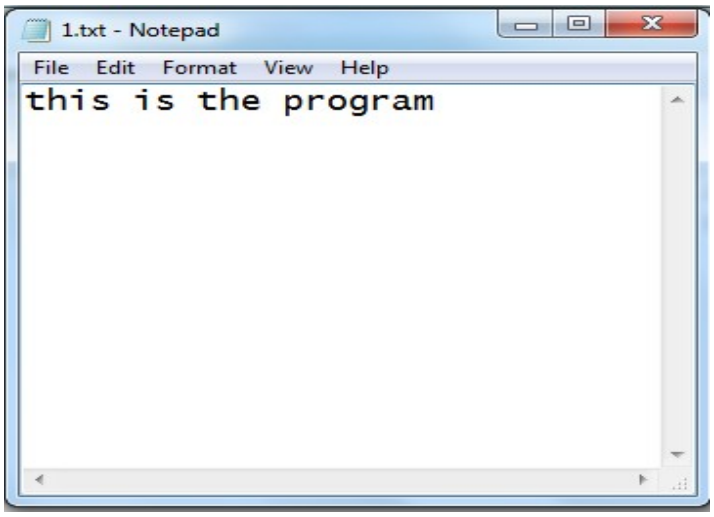


Practical 5

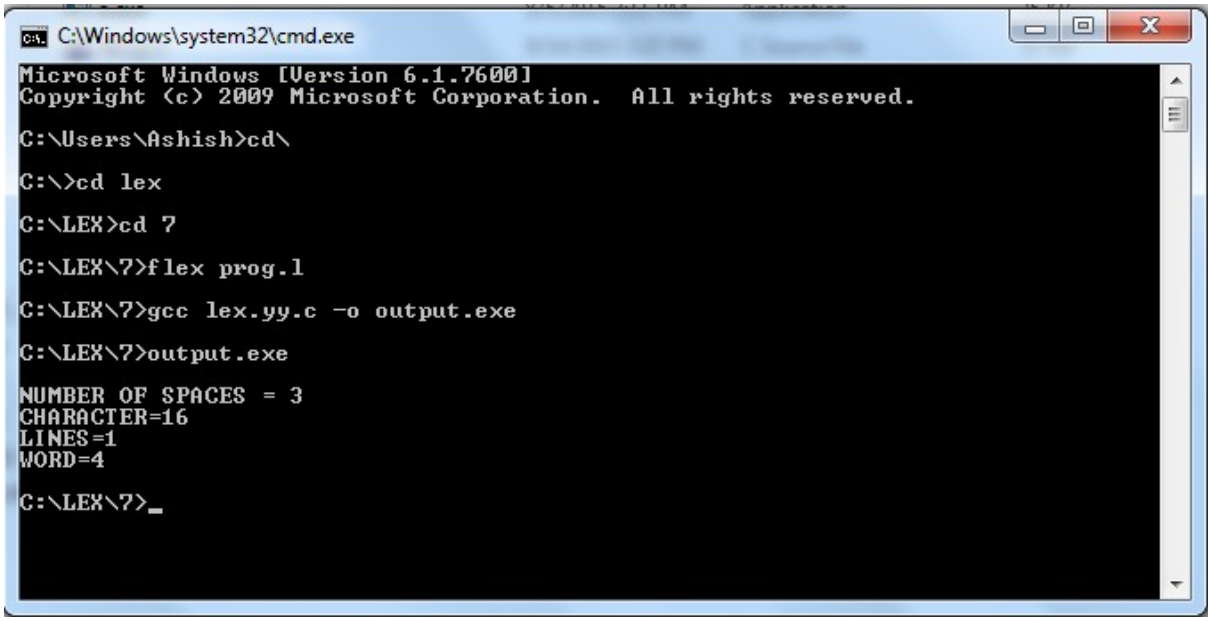
AIM: Write a Lex program to count the number of words, characters, blank spaces and lines.

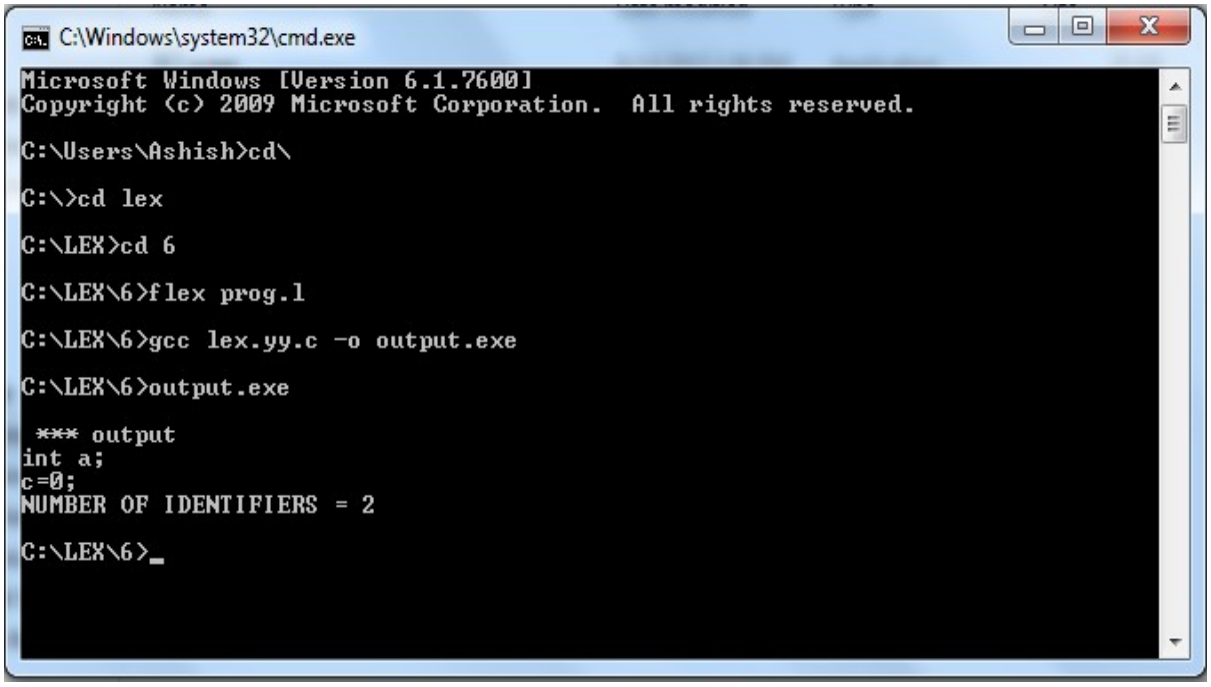
```
%{
int c=0,w=0,l=1,s=0;
%}
%%
[\n] {l++;}
[' '\n\t] {s++;}
[^' '\t\n]+ {w++; c+=yyleng;}
%%
int yywrap()
{
  return 1;
}
int main()
{
yyin=fopen("1.txt","r");
yylex();
printf("\nNUMBER OF SPACES = %d",s);
printf("\nCHARACTER=%d",c);
printf("\nLINES=%d",l);
printf("\nWORD=%d\n",w);
fclose(yyin);
}
```

INPUT FILE : `1.txt`



OUTPUT :


C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Ashish>cd\
C:\>cd lex
C:\LEX>cd 7
C:\LEX\7>flex prog.l
C:\LEX\7>gcc lex.yy.c -o output.exe
C:\LEX\7>output.exe
NUMBER OF SPACES = 3
CHARACTER=16
LINES=1
WORD=4
C:\LEX\7>


C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Ashish>cd\
C:\>cd lex
C:\LEX>cd 6
C:\LEX\6>flex prog.l
C:\LEX\6>gcc lex.yy.c -o output.exe
C:\LEX\6>output.exe
*** output
int a;
c=0;
NUMBER OF IDENTIFIERS = 2
C:\LEX\6>

Practical 6

AIM: Write a Program to generate Recursive Descent Parser.

ALGORITHM:

Step 1: start.

Step 2: Declare the prototype functions E() , EP(),T(), TP(),F()

Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals

Step 6: Read the parse string.

Step 7: stop the production

SOURCE CODE:

```
//RECURSIVE DECENT PARSER
#include<stdio.h>
#include<conio.h>
char *p;
E();
EP();
T();
F();
TP();
int main()
{
   clrscr();
   printf("The grammer is\n");
   printf("E->E+T/T\n");
   printf("T->T*F/F\n");
   printf("F->(E)/id\n");
   printf("\n Enter the string to be parsed:");
   scanf("%s",p);
   if(E())
       printf("\nString parsed");
   else
       printf("\nString not parsed");
   getch();
   return 0;
}
E()
{
   T();
   if(EP())
      return 1;
   else
      return 0;
}
EP()
{
   if(*p=='+')
   {
      p++;
      T();
   EP();
   }
```

```
   else if(*p=='\0')
      return 1;
   else
      return 0;
}
T()
{
  F();
  if(TP())
    return 1;
  else
     return 0;
}
TP()
{

   if(*p=='*')
   {
      p++;
      F();
      TP();
   }
    else if(*p=='\0')
      return 0;
}
F()
{
   if(*p=='i')
      p++;
   if(*p=='d')
      p++;
   else if(*p=='(')
     {
      p++;
      F();
      F();
     }
     else if(*p==')')
       p++;
      else
  return 0;
}
```

OUTPUT:

```
The grammar is
   E -> E+T/T
   T -> T*F/F
    F -> ( E )/id
Enter the string to be parsed
id + id * id
String parsed
```

Practical 7

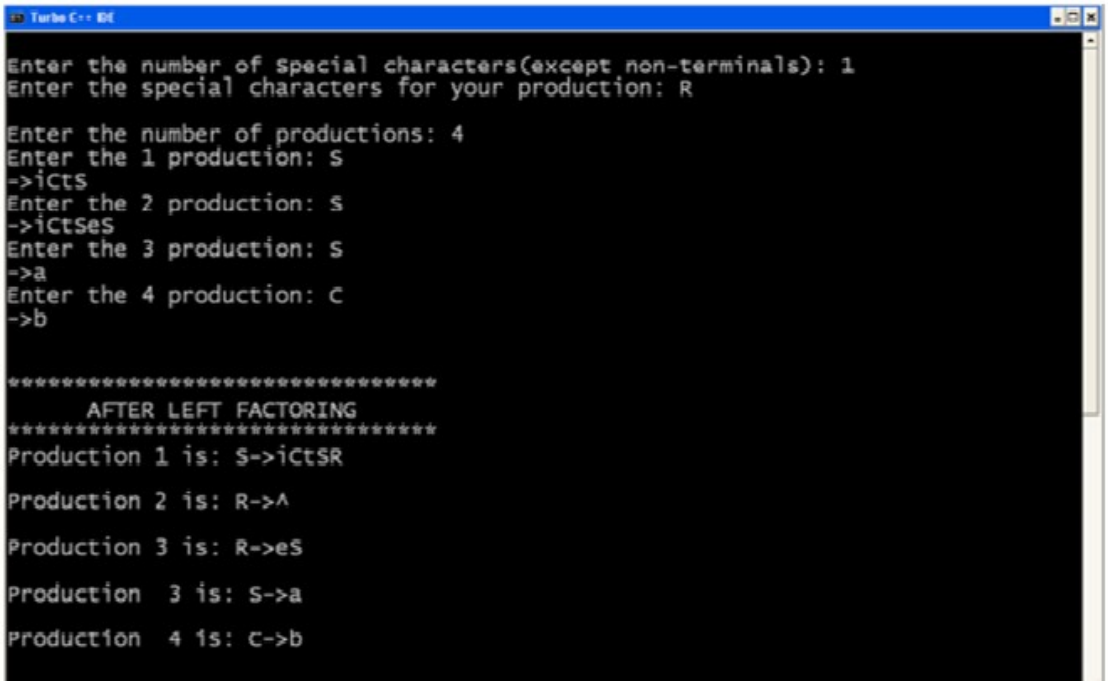
AIM: Write a program to left factor the given grammar.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

//Structure Declaration
struct production
{
            char lf;
            char rt[10];
            int prod_rear;
            int fl;
};
struct production prodn[20],prodn_new[20];     //Creation of object

//Variables Declaration
int b=-1,d,f,q,n,m=0,c=0;
char terminal[20],nonterm[20],alpha[10],extra[10];
char epsilon='^';

//Beginning of Main Program
void main()
{
   clrscr();

   //Input of Special characters
   cout<<"\nEnter the number of Special characters(except non-terminals): ";
   cin>>q;
   cout<<"Enter the special characters for your production: ";
   for(int cnt=0;cnt<q;cnt++)
   {
     cin>>alpha[cnt];
   }

   //Input of Productions
   cout<<"\nEnter the number of productions: ";
   cin>>n;
   for(cnt=0;cnt<=n-1;cnt++)
   {
     cout<<"Enter the "<< cnt+1<<" production: ";
     cin>>prodn[cnt].lf;
     cout<<"->";
     cin>>prodn[cnt].rt;
     prodn[cnt].prod_rear=strlen(prodn[cnt].rt);
     prodn[cnt].fl=0;
 }

   //Condition for left factoring
   for(int cnt1=0;cnt1<n;cnt1++)
   {
     for(int cnt2=cnt1+1;cnt2<n;cnt2++)
```

```
{
        if(prodn[cnt1].lf==prodn[cnt2].lf)
        {
          cnt=0;
          int p=-1;
          while((prodn[cnt1].rt[cnt]!='\0')&&(prodn[cnt2].rt[cnt]!='\0'))
          {
            if(prodn[cnt1].rt[cnt]==prodn[cnt2].rt[cnt])
            {
              extra[++p]=prodn[cnt1].rt[cnt];
              prodn[cnt1].fl=1;
              prodn[cnt2].fl=1;
            }
            else
            {
              if(p==-1)
                      break;
              else
              {
                      int h=0,u=0;
                      prodn_new[++b].lf=prodn[cnt1].lf;
                      strcpy(prodn_new[b].rt,extra);
                      prodn_new[b].rt[p+1]=alpha[c];
                      prodn_new[++b].lf=alpha[c];
                      for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
                       prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
                       prodn_new[++b].lf=alpha[c];
                      for(g=cnt;g<=prodn[cnt1].prod_rear;g++)
                       prodn_new[b].rt[u++]=prodn[cnt1].rt[g];
                       m=1;
                       break;
              }
             }
            cnt++;
          }
          if((prodn[cnt1].rt[cnt]==0)&&(m==0))
          {
                    int h=0;
                    prodn_new[++b].lf=prodn[cnt1].lf;
                    strcpy(prodn_new[b].rt,extra);
                    prodn_new[b].rt[p+1]=alpha[c];
                    prodn_new[++b].lf=alpha[c];
                    prodn_new[b].rt[0]=epsilon;
         prodn_new[++b].lf=alpha[c];
               for(int g=cnt;g<prodn[cnt2].prod_rear;g++)
                     prodn_new[b].rt[h++]=prodn[cnt2].rt[g];
          }
          if((prodn[cnt2].rt[cnt]==0)&&(m==0))
          {
              int h=0;
              prodn_new[++b].lf=prodn[cnt1].lf;
              strcpy(prodn_new[b].rt,extra);
              prodn_new[b].rt[p+1]=alpha[c];
              prodn_new[++b].lf=alpha[c];
              prodn_new[b].rt[0]=epsilon;
              prodn_new[++b].lf=alpha[c];
```

```
                    for(int g=cnt;g<prodn[cnt1].prod_rear;g++)
                    prodn_new[b].rt[h++]=prodn[cnt1].rt[g];
                }
                c++;
                m=0;
             }
        }
     }

   //Display of Output
     cout<<"\n\n*********************************";
     cout<<"\n      AFTER LEFT FACTORING      ";
     cout<<"\n*********************************";
     cout<<endl;
     for(int cnt3=0;cnt3<=b;cnt3++)
             {
                         cout<<"Production "<<cnt3+1<<" is: ";
                         cout<<prodn_new[cnt3].lf;
                         cout<<"->";
                         cout<<prodn_new[cnt3].rt;
                         cout<<endl<<endl;
             }

     for(int cnt4=0;cnt4<n;cnt4++)
     {
     if(prodn[cnt4].fl==0)
     {
     cout<<"Production  "<<cnt3++<<" is: ";
     cout<<prodn[cnt4].lf;
     cout<<"->";
     cout<<prodn[cnt4].rt;
     cout<<endl<<endl;
     }
   }
  getch();
}    //end of main program
```

OUTPUT:


Turbo C++ IDE
Enter the number of special characters(except non-terminals): 1
Enter the special characters for your production: R
Enter the number of productions: 4
Enter the 1 production: S
->iCtS
Enter the 2 production: S
->iCtSeS
Enter the 3 production: S
->a
Enter the 4 production: C
->b
****
AFTER LEFT FACTORING
****
Production 1 is: S->iCtSR
Production 2 is: R->^
Production 3 is: R->eS
Production 3 is: S->a
Production 4 is: C->b

Practical 8

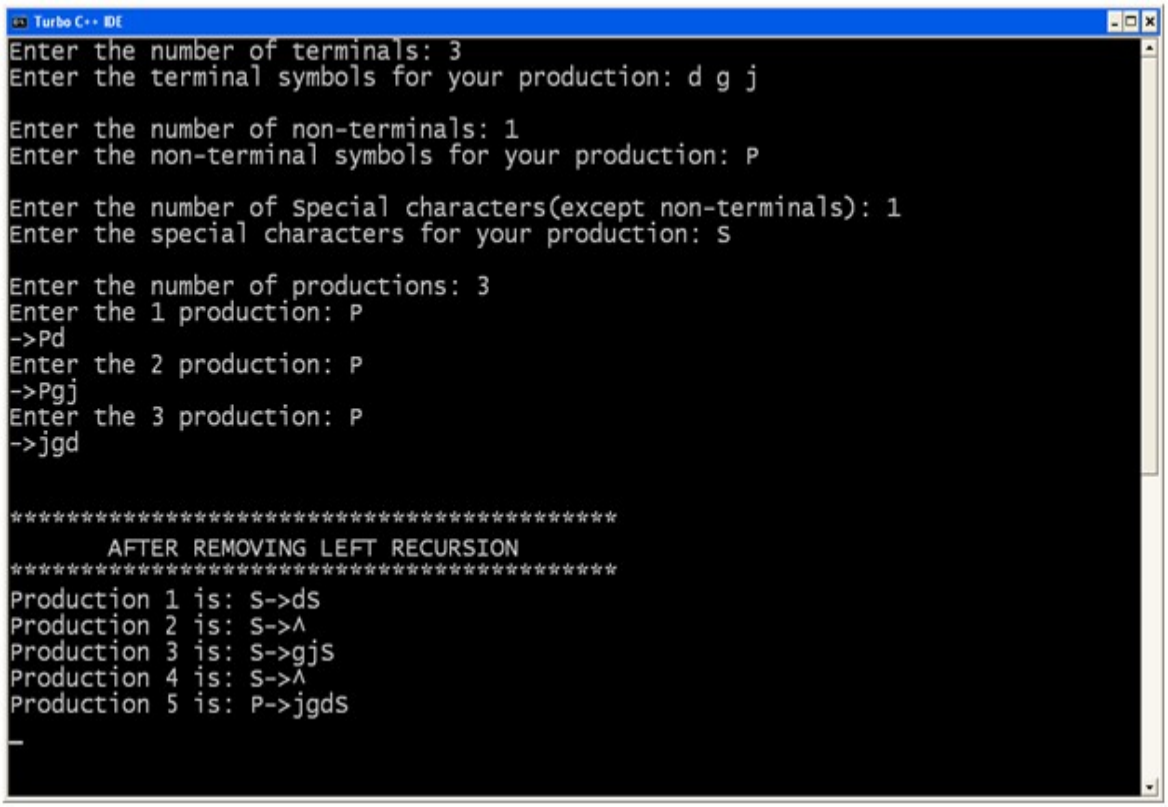
AIM: Write a program to remove the Left Recursion from a given grammar.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

struct production
{
            char l;
            char r[10];
            int rear;
};
struct production prod[20],pr_new[20];

int p=0,b=0,d,f,q,n,flag=0;
char terminal[20],nonterm[20],alpha[10];
char x,epsilon='^';

void main()
{
    clrscr();

       cout<<"Enter the number of terminals: ";
    cin>>d;
    cout<<"Enter the terminal symbols for your production: ";
    for(int k=0;k<d;k++)
    {
      cin>>terminal[k];
    }


    cout<<"\nEnter the number of non-terminals: ";
    cin>>f;
    cout<<"Enter the non-terminal symbols for your production: ";
    for(k=0;k<f;k++)
    {
      cin>>nonterm[k];
    }


    cout<<"\nEnter the number of Special characters(except non-terminals): ";
    cin>>q;
    cout<<"Enter the special characters for your production: ";
    for(k=0;k<q;k++)
    {
      cin>>alpha[k];
    }


    cout<<"\nEnter the number of productions: ";
  cin>>n;
    for(k=0;k<=n-1;k++)
    {
      cout<<"Enter the "<< k+1<<" production: ";
```

```
    cin>>prod[k].l;
    cout<<"->";
    cin>>prod[k].r;
    prod[k].rear=strlen(prod[k].r);
  }


  for(int m=0;m<f;m++)
  {
    x=nonterm[m];
    for(int j=0;j<n;j++)
    {
            if((prod[j].l==x)&&(prod[j].r[0]==prod[j].l))
               flag=1;
    }
    for(int i=0;i<n;i++)
    {
            if((prod[i].l==x)&&(prod[i].r[0]!=x)&&(flag==1))
            {
                        pr_new[b].l=x;
                        for(int c=0;c<prod[i].rear;c++)
                        pr_new[b].r[c]=prod[i].r[c];
                        pr_new[b++].r[c]=alpha[p];
            }
            else if((prod[i].l==x)&&(prod[i].r[0]==x)&&(flag==1))
            {
                       pr_new[b].l=alpha[p];
                       for(int a=0;a<=prod[i].rear-2;a++)
                        pr_new[b].r[a]=prod[i].r[a+1];
                       pr_new[b++].r[a]=alpha[p];
                       pr_new[b].l=alpha[p];
                       pr_new[b++].r[0]=epsilon;
             }
             else if((prod[i].l==x)&&(prod[i].r[0]!=x)&&(flag==0))
             {
                        pr_new[b].l=prod[i].l;
                        strcpy(pr_new[b].r,prod[i].r);
                        b++;
             }
          }
          flag=0;
          p++;
}


 cout<<"\n\n*****************************************";
  cout<<"\n       AFTER REMOVING LEFT RECURSION       ";
  cout<<"\n*****************************************"<<endl;
  for(int s=0;s<=b-1;s++)
          {
                     cout<<"Production "<<s+1<<" is: ";
                     cout<<pr_new[s].l;
                     cout<<"->";
                     cout<<pr_new[s].r;
                     cout<<endl;
          }
```

```
    getch();
}
```

OUTPUT:


Turbo C++ IDE
Enter the number of terminals: 3
Enter the terminal symbols for your production: d g j
Enter the number of non-terminals: 1
Enter the non-terminal symbols for your production: P
Enter the number of Special characters(except non-terminals): 1
Enter the special characters for your production: S
Enter the number of productions: 3
Enter the 1 production: P
->Pd
Enter the 2 production: P
->Pgj
Enter the 3 production: P
->jgd
AFTER REMOVING LEFT RECURSION
Production 1 is: S->dS
Production 2 is: S->^
Production 3 is: S->gjS
Production 4 is: S->^
Production 5 is: P->jgdS

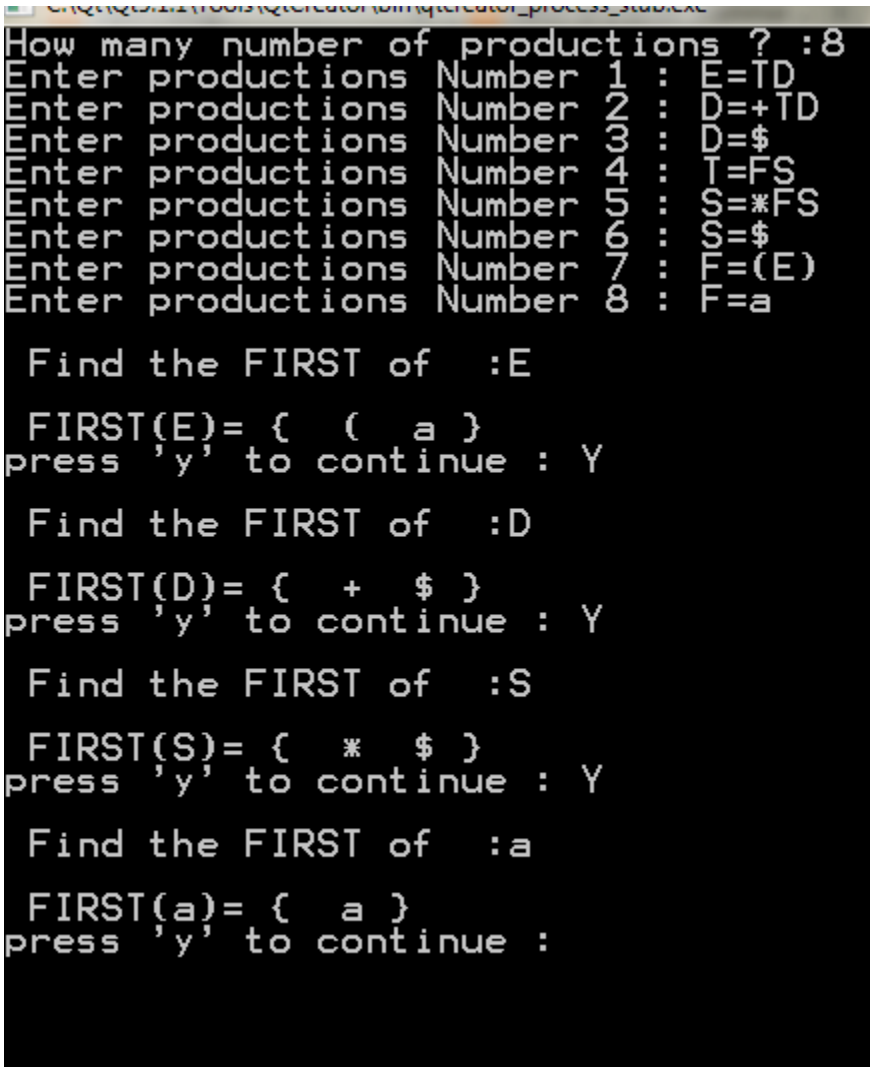
Practical 9

AIM: Write a program to find the FIRST set of the given grammar.

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);
    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of  :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)
        printf(" %c ",result[i]);       //Display result
        printf("}\n");
         printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
    while(choice=='y'||choice =='Y');
}
/*
 *Function FIRST:
 *Compute the elements in FIRST(c) and write them
 *in Result Array.
 */
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
    //If X is terminal, FIRST(X) = {X}.
    if(!(isupper(c)))
    {
        addToResultSet(Result,c);
               return ;
  }
    //If X is non terminal
    //Read each production
```

```
    for(i=0;i<numOfProductions;i++)
    {
//Find production with X as LHS
        if(productionSet[i][0]==c)
        {
//If X → ε is a production, then add ε to FIRST(X).
 if(productionSet[i][2]=='$') addToResultSet(Result,'$');
            //If X is a non-terminal, and X → Y1 Y2 … Yk
            //is a production, then add a to FIRST(X)
            //if for some i, a is in FIRST(Yi),
            //and ε is in all of FIRST(Y1), …, FIRST(Yi-1).
      else
            {
                j=2;
                while(productionSet[i][j]!='\0')
                {
                foundEpsilon=0;
                FIRST(subResult,productionSet[i][j]);
                for(k=0;subResult[k]!='\0';k++)
                    addToResultSet(Result,subResult[k]);
                 for(k=0;subResult[k]!='\0';k++)
                     if(subResult[k]=='$')
                     {
                         foundEpsilon=1;
                         break;
                     }
                 //No ε found, no need to check next element
                 if(!foundEpsilon)
                     break;
                 j++;
                }
            }
    }
}
    return ;
}
/* addToResultSet adds the computed
 *element to result set.
 *This code avoids multiple inclusion of elements
  */
void addToResultSet(char Result[],char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
  Result[k+1]='\0';
}
```

OUTPUT:


How many number of productions ? :8
Enter productions Number 1 : E=TD
Enter productions Number 2 : D=+TD
Enter productions Number 3 : D=$
Enter productions Number 4 : T=FS
Enter productions Number 5 : S=*FS
Enter productions Number 6 : S=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=a
Find the FIRST of :E
FIRST(E)= { (a }
press 'y' to continue : Y
Find the FIRST of :D
FIRST(D)= { + $ }
press 'y' to continue : Y
Find the FIRST of :S
FIRST(S)= { * $ }
press 'y' to continue : Y
Find the FIRST of :a
FIRST(a)= { a }
press 'y' to continue :

Practical 10

AIM: Write a program to find the FOLLOW set of the given grammar.

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],followResult[10];
void follow(char c);
void first(char c);
void addToResult(char);
int main()
{
 int i;
 int choice;
 char c,ch;
 printf("Enter the no.of productions: ");
scanf("%d", &n);
 printf(" Enter %d productions\nProduction with multiple terms should be give as
separate productions \n", n);
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);
    // gets(a[i]);
 do
 {
  m=0;
  printf("Find FOLLOW of -->");
  scanf(" %c",&c);
  follow(c);
  printf("FOLLOW(%c) = { ",c);
  for(i=0;i<m;i++)
   printf("%c ",followResult[i]);
  printf(" }\n");
  printf("Do you want to continue(Press 1 to continue....)?");
 scanf("%d%c",&choice,&ch);
 }
 while(choice==1);
}
void follow(char c)
{
    if(a[0][0]==c)addToResult('$');
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);
    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);
   }
  }
 }
}
void first(char c)
```

```
{
      int k;
                  if(!(isupper(c)))
                      //f[m++]=c;
                      addToResult(c);
                  for(k=0;k<n;k++)
                  {
                  if(a[k][0]==c)
                  {
                  if(a[k][2]=='$') follow(a[i][0]);
                  else if(islower(a[k][2]))
                      //f[m++]=a[k][2];
                      addToResult(a[k][2]);
                  else first(a[k][2]);
                  }
                  }
}
void  addToResult(char c)
{
    int i;
    for( i=0;i<=m;i++)
        if(followResult[i]==c)
            return;
   followResult[m++]=c;
}
```

OUTPUT:

```
Enter the no.of productions: 8
 Enter 8 productions
Production with multiple terms should be give as separate productions
E=TD
D=+TD
D=$
T=FS
S=*FS
S=$
F=(E)
F=a
Find FOLLOW of -->E
FOLLOW(E) = { $ )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->D
FOLLOW(D) = { )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->T
FOLLOW(T) = { + $ )  }
Do you want to continue(Press 1 to continue....)?S
Find FOLLOW of -->FOLLOW(S) = { $ )  }
Do you want to continue(Press 1 to continue....)?1
Find FOLLOW of -->F
FOLLOW(F) = { * + $ )  }
Do you want to continue(Press 1 to continue....)?
```

Practical 11

AIM: Program to generate Parsing Table for the Predictive Parser.

```
#include<conio.h>
#include<stdio.h>
#include<math.h>

char g[5][2][10]={
                    {"e->tE"},          //treat E as e'
                    {{"E->+tE"},{"E->N"}},   //treat N as null
                    {"t->fT"},              //treat T as t'
                    {{"T->*fT"},{"T->N"}},
                    {{"f->(e)"},{"f->i"}},
             };
char resf[5][2];
char resfollow[5][5];
int track[5];
char table[5][6][7];
char ind_nt[]={'i','+','*','(',')','$'};

void ptable(void);
void first(char);
void follow(void);

void main()
{
       int i,k;
       clrscr();
       //find first
       for(k=0;k<5;k++)
       {
              for(i=0;i<5;i++)
                     track[i]=0;
              if(resf[k][0]=='\x0')
                     first(g[k][0][0]);
       }
       follow();
       ptable();
       for(i=0;i<6;i++)
              printf("%10c",ind_nt[i]);
       printf("\n\n");

       for(i=0;i<5;i++)
       {
              printf("%c",ind_nt[i]);
              for(k=0;k<6;k++)
              {
              printf(" %10s",table[i][k]);
              }
              printf("\n\n");
       }
       getch();
}
```

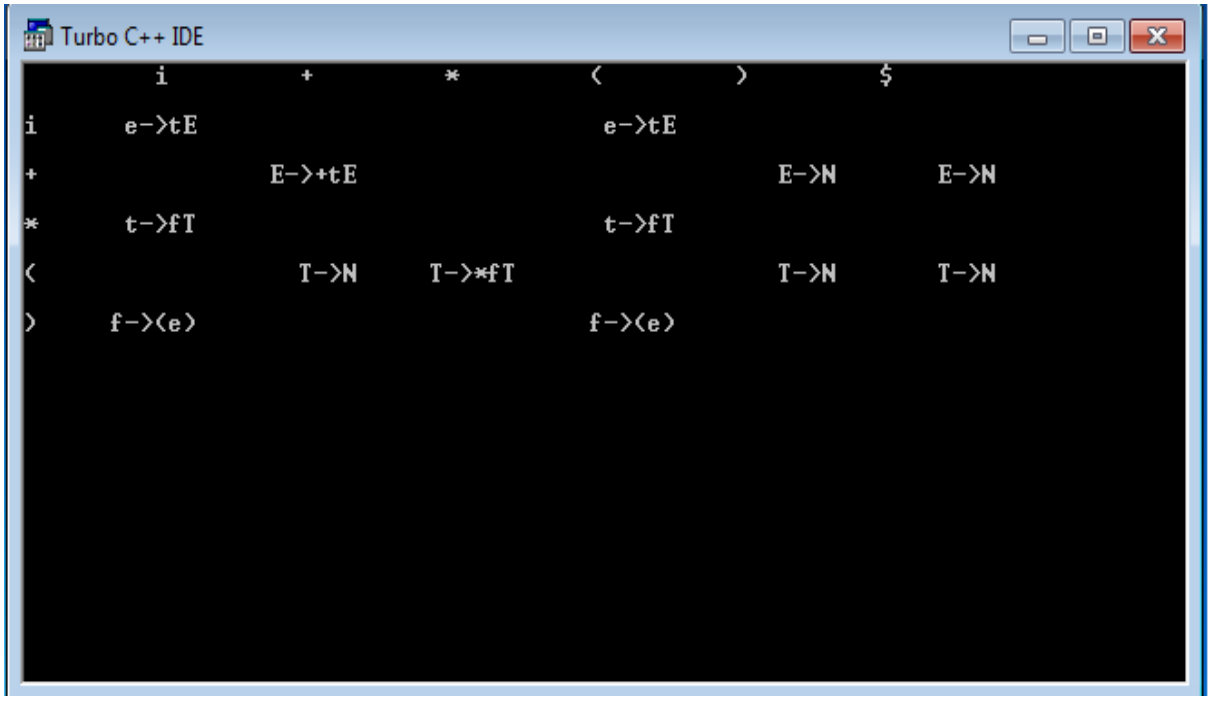
```
void first(char ref)
{
	int i,j,production,k;
	char c;
	for(i=0;i<5;i++)
	{
		if(g[i][0][0]==ref)
		{
			production=i;
			track[production]=1;
		}
	}
	for(i=0;i<2;i++)
	{
		c=g[production][i][0+3];
		if(c=='+'||c=='*'||c=='('||c==')'||c=='i'||c=='N')
		{
			for(k=0;k<5;k++)
			{
				if(track[k]!=0)
					resf[k][i]=c;
			}
		}
		else if(c !='\x0')

			first(c);
	}
}
void follow(void)
{
	int i,j,k,l,pbeta,pB,pA;
	int index[5]={0,0,0,0,0};
	int iA,iB,ibeta;
	int rule3;
	int length,redundant;
	char start,a,c,B,beta,A;
	//rule 1
	for(i=0;i<5;i++)
	{
		for(j=0;j<2;j++)
		{
			start=g[i][j][3];
			if(start != '+' && start != '*' && start != '(' && start !=
')' && start != 'N' && start != 'i' && start!='\x0')
				resfollow[i][index[i]++]='$';
		}
		//defination
		length=strlen(g[i][0]);
		a=g[i][0][length-1];
		A=g[i][0][length-2];
		if((a=='+'||a=='*'||a=='('||a==')'||a=='i')&&a!='N')
		{
		//search for the index value of A
			for(k=0;k<5;k++)
				if(g[k][0][0]==A)
					pA=k;
			resfollow[pA][index[pA]++]=a;
```

```
			}
		}
		//rule 2
		for(j=0;j<5;j++)
		{
			A=g[j][0][0];
			length=strlen(g[j][0]);
			B=g[j][0][length-2];
			beta=g[j][0][length-1];
			for(i=0;i<5;i++)
			{
				if(g[i][0][0]==beta)
					pbeta=i;
				if(g[i][0][0]==A)
					pA=i;
				if(g[i][0][0]==B)
					pB=i;
			}
			if(!(beta=='+' || beta=='*' || beta=='(' || beta==')' || beta=='i')
&& beta !='N')
			{
				for(i=0;i<2;i++)
				{
					if((resf[pbeta][i])!='N')
					{
						//check for redundant element in follow
						redundant=0;
						for(k=0;k<5;k++)
						{
							if(resfollow[pB][k]== resf[pbeta][i])
								redundant=1;
						}
						if(redundant!=1)
							(resfollow[pB][index[pB]++])=(resf[pbeta]
[i]);
					}
				}
			}
		}
		//rule 3 A->(alpha)(B)(beta)
		for(j=0;j<5;j++)
		{
			A=g[j][0][0];
			length=strlen(g[j][0]);
			B=g[j][0][length-2];
			beta=g[j][0][length-1];
			if(!(beta=='+'||beta=='*'||beta=='('||beta==')'||beta=='i'||
beta=='N'))
			{
				for(i=0;i<5;i++)
				{
					if(g[i][0][0]==beta)
						pbeta=i;
					if(g[i][0][0]==A)
						pA=i;
					if(g[i][0][0]==B)
						pB=i;
```

```
                    }
                    for(i=0;i<2;i++)
                    {
                           if(resf[pbeta][i]=='N')
                           {
                                  for(k=0;k<5;k++)
                                  {
                                         //check for redundant element in follow
                                         redundant=0;
                                         for(l=0;l<5;l++)
                                         {
                                         if(resfollow[pB][l]== resfollow[pA][k])
                                                redundant=1;
                                         }
                                         if(redundant!=1)
                                                (resfollow[pB][index[pB]+
+])=(resfollow[pA][k]);
                                  }

                           }

                    }
             }
      }
      //rule 3 A->(alpha)(B)
      for(j=0;j<5;j++)
      {
             A=g[j][0][0];
             length=strlen(g[j][0]);
             B=g[j][0][length-1];
             if(A!=B)
             {
                    for(i=0;i<5;i++)
                    {
                           if(g[i][0][0]==A)
                                  pA=i;
                           if(g[i][0][0]==B)
                                  pB=i;
                    }
                    if(!(B=='+'||B=='*'||B=='('||B==')'||B=='i'||B=='N'))
                           for(i=0;i<5;i++)
                                  resfollow[pB][index[pB]++]=resfollow[pA][i];
             }
      }
}
void ptable(void)
{
      int i,j,k,l,m;
      char A;
      for(i=0;i<5;i++)
      {
             A=g[i][0][0];
             for(j=0;j<2;j++)
             {
                    //step 2
                    if(resf[i][j]!='N')
                    {
```

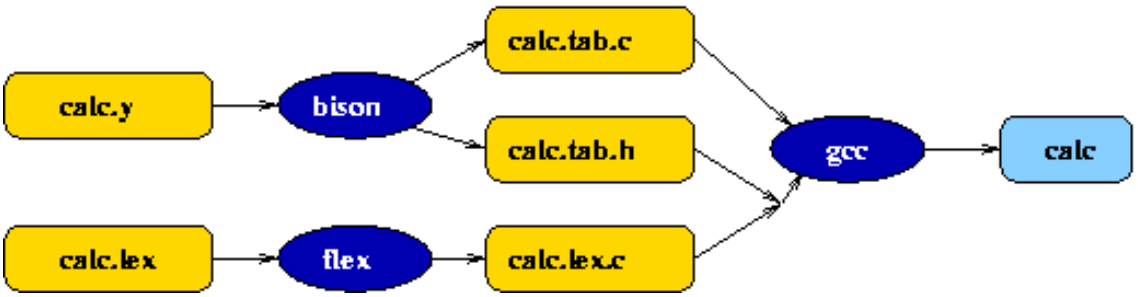
```
                        for(k=0;k<6;k++)
                              if(ind_nt[k]==resf[i][j])
                                    strcpy(table[i][k],g[i][0]);
                  }
            }
            //step 3
            for(j=0;j<2;j++)
            {
                  if(resf[i][j]=='N')
                  {
                        for(k=0;k<3;k++)
                        {
                              for(l=0;l<6;l++)
                              {
                                    if(ind_nt[l]==resfollow[i][k])
                                          strcpy(table[i][l],g[i][1]);
                              }
                        }
                  }
            }
      }
}
```

OUTPUT :



Practical 12

AIM: Program to implement Higher Order Calculator 1 (HOC1) by using YACC parser generator.



The aim of this example is to write a calculator that will understand +, -, *, /, and ^ (power) and give them the correct precedence, understanding brackets. Numbers can be integers, or float.

calc.y:

```
%{
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define YYSTYPE double
%}

%token NUMBER
%token PLUS MINUS TIMES DIVIDE POWER
%token LEFT RIGHT
%token END

%left PLUS MINUS
%left TIMES DIVIDE
%left NEG
%right POWER

%start Input
%%

Input:

     | Input Line
;

Line:
     END
     | Expression END { printf("Result: %f\n", $1); }
;

Expression:
     NUMBER { $$=$1; }
```

```
| Expression PLUS Expression { $$=$1+$3; }
| Expression MINUS Expression { $$=$1-$3; }
| Expression TIMES Expression { $$=$1*$3; }
| Expression DIVIDE Expression { $$=$1/$3; }
| MINUS Expression %prec NEG { $$=-$2; }
| Expression POWER Expression { $$=pow($1,$3); }
| LEFT Expression RIGHT { $$=$2; }
;

%%

int yyerror(char *s) {
  printf("%s\n", s);
}

int main() {
 if (yyparse())
     fprintf(stderr, "Successful parsing.\n");
  else
     fprintf(stderr, "error found.\n");
}
```

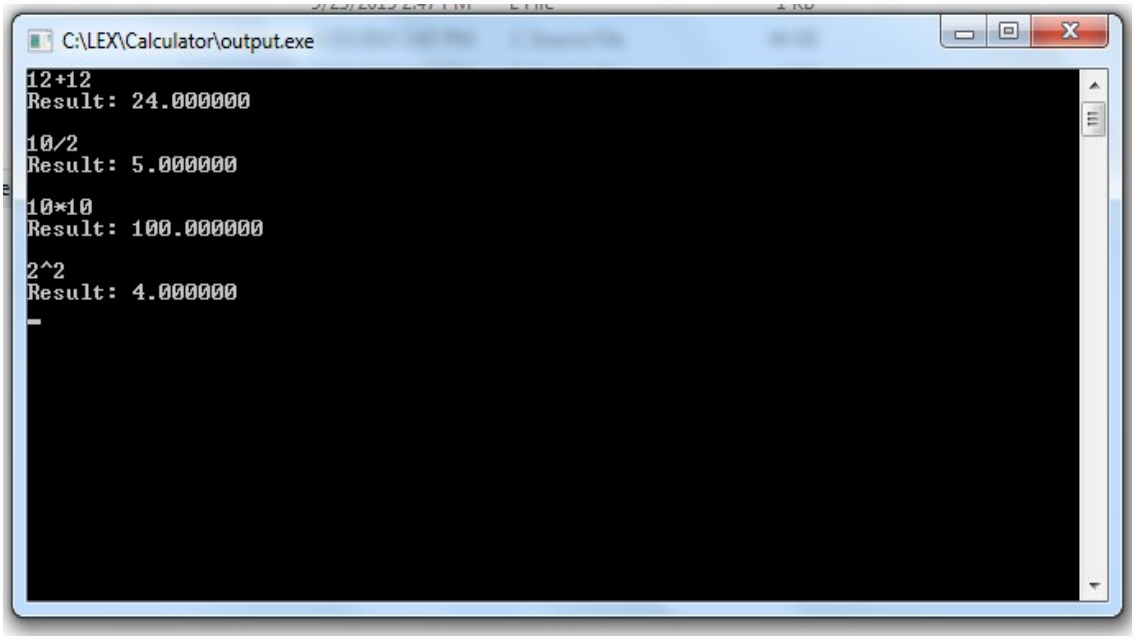
calc.l:

```
%{
#define YYSTYPE double
#include "calc.tab.h"
#include <stdlib.h>
%}

white [ \t]+
digit [0-9]
integer {digit}+
exponent [eE][+-]?{integer}
real {integer}("."{integer})?{exponent}?


%%
{white} { }
{real} { yylval=atof(yytext); return NUMBER; }

"+" return PLUS;
"-" return MINUS;
"*" return TIMES;
"/" return DIVIDE;
"^" return POWER;
"(" return LEFT;
")" return RIGHT;
"\n" return END;

%%

int yywrap()
{
       return 1;
}
```

Final Output :

```
>gcc lex.yy.c calc.tab.c -o output.exe
```

1) Using above command output executable file will be generated as given below.



2) Here is an example of incorrect syntax and hence parser giving error message.